Web Components: Developer Happiness Through Security and Usability

Christopher Helle and Teemu Vartiainen
School of Science, Aalto University
{christopher.helle,teemu.t.vartiainen}@aalto.fi

Abstract – **In this paper we introduce a general overview of the current status of the Web Components standards. We introduce the technologies and discuss the related work. Further, we discuss the main benefits of the technologies and show some basic use cases. Web Components provide a native way of building modular pages without external JavaScript frameworks and thus it will be more stable and future-proof than the alternatives. Web components still only have partial native browser support but with the help of Polyfill library, features work as intended on all modern browsers.**

Keywords – *Web Components, JavaScript, client-side templates, standards*

1 INTRODUCTION

The Web development community has developed a vast amount of technologies to succeed in a fairly innocent task: providing the developer basic components for building user interfaces. The TodoMVC project[1] lists a rainbow of frameworks for a developer to choose from. However, using a framework has its drawbacks. They are a hard dependency for the system, they have strong “flavor of the month” tendencies and they lack interoperability [1]. Solving these problems would require a wide agreement among the framework developers which seems like an infeasible task.

A solution for this has been proposed by W3C, the standardizing organization for the Web. W3C is working on defining “Web Components”, a collection of standards for defining custom, reusable, and native building blocks for the Web. There are four key technologies involved in Web Components: Custom elements [2], HTML Templates [3], Shadow DOM [4], and HTML Import [5].

1.1 CUSTOM ELEMENTS

The new standards [2] allow developers to define new custom HTML tags with a simple JavaScript call (see Figure 1). It allows developers to bundle together logical functionalities under single tag or extend external API functionalities or components. The developer can register callbacks to listen for events such as attribute changes on the element.

```
var MyElement =
  document.registerElement('my-element', {
    prototype: Object.create(HTMLElement.prototype)
  });
```

Figure 1: New element registration

1.2 HTML TEMPLATES

Many custom elements such as buttons will need a user interface component to be included. The custom element developer should be able to design the UI for the element with the existing native Web technologies.

Client side templates is one of the biggest benefit provided by the multitude of Web frameworks. However, defining a truly invisible template for an element that doesn’t have side effects on the main document has been a challenge. The new `<template>`-element in HTML5 [3] provides a standardized DOM-based approach for client-side templating. It allows the developer to specify a cloneable chunk of HTML that is inert and won’t be rendered until activated.

1.3 SHADOW DOM

A HTML webpage consists of a DOM-tree (Document Object Model) which is a tree structure containing all the nodes (elements) of the webpage. This tree structure is used to render the page and it can be addressed and manipulated from any JavaScript code. A downside of this public addressing is that any code running anywhere on the page can access the whole DOM-tree.

Shadow DOM aims to address this public addressability by defining a way of inserting a separate “Shadow” tree for a node in the public DOM [6]. The browser renders the whole tree combining the main document and the shadow trees. However, the shadow trees are encapsulated and invisible. For any scripts they appear as a single node.

In addition to hiding the shadow tree from scripts, the encapsulation also makes sure that any styles or scripts from inside the Shadow DOM are encapsulated inside the shadow tree [1].

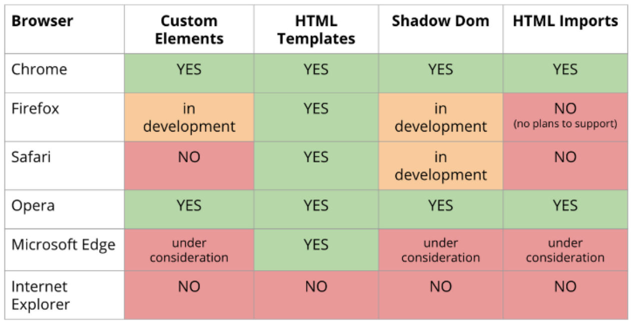
[1] http://todomvc.com/

1.4 HTML Import

A key aspect of Web Components is reusability. With a HTML Import [5] the developer can include other HTML files into the current document. The imported file can include all the three previously presented technologies to bring a readily packaged custom element onto the page. This will also allow the browser to cache and de-duplicate the contents [1]. This way the developer can immediately use the new custom tag without worrying about other content on the page. In practice using the import works identically to importing CSS stylesheets.

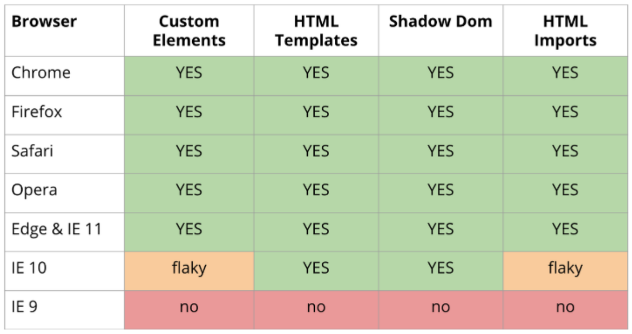
2 Related work

Currently native browser compatibility for web components is limited. Only Google Chrome and Opera browsers have native support for all four main technologies: Shadow DOM, HTML import, HTML templates and custom elements[2]. Out of these only HTML templates are currently supported on all latest browsers including Microsoft's Edge's latest version. Support for other browsers or older browser versions can be extended by using externally loadable webcomponents.js polyfill library so that web components functionality can be used on almost all modern platforms.



Browser	Custom Elements	HTML Templates	Shadow Dom	HTML Imports
Chrome	YES	YES	YES	YES
Firefox	in development	YES	in development	NO (no plans to support)
Safari	NO	YES	in development	NO
Opera	YES	YES	YES	YES
Microsoft Edge	under consideration	YES	under consideration	under consideration
Internet Explorer	NO	NO	NO	NO

Table 1: Current status of native browser support



Browser	Custom Elements	HTML Templates	Shadow Dom	HTML Imports
Chrome	YES	YES	YES	YES
Firefox	YES	YES	YES	YES
Safari	YES	YES	YES	YES
Opera	YES	YES	YES	YES
Edge & IE 11	YES	YES	YES	YES
IE 10	flaky	YES	YES	flaky
IE 9	no	no	no	no

Table 2: Browser support with polyfill

Firefox has no plans to support HTML imports though for now it can be enabled through the "dom.webcomponents.enabled" preference in about:config [7]. The browser developers state that due to the current volatile state of new web frameworks they aim to avoid following a new standard before it has been stabilized with incoming technologies such as JavaScript modules [8]. Also, with the functionality of polyfill libraries, the need for native implementation of HTML imports is not that acute.

It seems like Web Components are still emerging technologies and not used extensively in production environments. However, there are some public big projects that use partially or extensively Web components features.

Maybe the biggest project around Web components is Google Polymer library aiming to provide small high quality particles of a web page ready to be embedded and used in someone's web design. Polymer reached production ready release 1.0 on 28th May 2015[3] with a message that it's stable and should work on most major modern browsers with the help of polyfill library. The Polymer library has extensive collection of elements that follow Google's own material design theme guide used on Android systems and the library seems pretty and fluid.

Also Google's popular front end JavaScript framework Angular relies in it's coming 2.0 version on Shadow DOM functionality in browser. The specifications are not yet clear. Angular 1.x is one of the most popular front end JavaScript frameworks and the release for a renewed version 2.0 has gathered lots of expectations.

It seems GitHub has adopted custom elements functionality in their popular project collaboration and version control tool[4]. Developer Peek [8] says that most of the web components features are still too cutting edge and unstable to be implemented in their service but using the custom elements makes the markup more semantic.

3 Benefits

The benefits can be categorized into three major categories.

3.1 Developer's benefits

Savage [1] describes how current status of web development is revolving around different frameworks. He states that the frameworks are not compatible between each other and their usage patterns are so different that switching a framework is like learning a new language; it requires a huge effort to adapt to something else.

Savage also suggested that the web components provide useful tools for web development that do not rely on some external frameworks but the universal browser functionalities. Web component implementations are

[2] caniuse.com, 12. November 2015

[3] https://blog.polymer-project.org/announcements/2015/05/29/one-dot-oh/

[4] http://github.com

more versatile and adaptable to different environments and makes web development more pleasant for the developer.

3.1.1 Semantical informative syntax

Custom elements and Shadow DOM enable the developer to use more semantic HTML syntax. Currently complex pages require lots of div elements to be able to implement desired styling. The HTML markup for such page is not so easy to understand what each element does. With Custom elements it is possible to define new elements to describe the content in a more semantic way. Furthermore, it is possible to hide all the asemantic elements from the DOM by encapsulating the element into Shadow DOM. When the elements related to the styling and visuals are hidden in Shadow DOM the HTML markup can focus on what is important: content and semantics.

In chapter 5 we demonstrate a practical example on creating a Shadow DOM encapsulation to separate content from styling.

3.1.2 Modular thinking & multiplication

Web components encourage modular thinking. Web pages may be developed in smaller components and then imported to the main page. This approach differs from the current method of building a page as a single component.

With web component functionalities it's possible to create custom components that can be multiplied or inserted anywhere and the developer may trust that it looks the same because of the component isolation. With the shadow DOM [3] the loaded templates can be isolated so that external CSS cannot break the styling of the template. The three functionalities HTML templates, Shadow DOM and HTML import work very well together to make the HTML structure more dynamic.

Since the components act as independent isolated blocks and are not so reliant on the pages CSS styling or JavaScript they can be easily reused in different context. If some buttons or page elements are used in a project later on they may be used in another project as well. This is a major improvement to the current situation; when starting a project on another framework basically not much can be utilized from the previous work. Web components that rely on HTML specifications and are implemented in a browser do not care about external frameworks as Web Components work in any environment.

3.2 Encapsulation and security

Third-party libraries for user interface components include scripts and styling. Unless both the developer of the library and the developer using the library use carefully scoped variable and class names, there is a chance that the names chosen overlap. For example, using a simple common name like "button" for styling can be harmful. This may end up with code from the library overlapping with the developer's own code causing unexpected results.

As stated earlier, Web Components and Shadow DOM address this by encapsulating styles and scripts inside the shadow tree. This way the developer can be sure there are no conflicts between the main document and the different libraries. A style rule used in a Web Component is guaranteed to stay inside the component's shadow tree. The developer can also trust that the scripts inside the component will not affect the main document.

Web Components can also be exploited for security-critical applications. ShadowCrypt [6] uses the Shadow DOM to display encrypted information on the webpage. The information (in this case, text) is stored inside the Shadow DOM and with the library it is impossible for another untrusted script on the webpage to access the content of that text. However, since the browser renders a composite of the main DOM tree and the shadow trees, the user sees the text as a regular part of the page.

3.3 Performance

With HTML imports all the necessary code and further dependencies for a Web Component is stored in a single file separate from the main document. This allows the file to be cached by the browser thus removing the need for downloading the code on every page load.

4 Critique

A downside of Web Components is that they transfer the responsibility of following conventions to the developer of the component. For example, accessibility features such as tags and classification for screen readers are built in to the default elements [1]. However, for custom elements adding such accessibility features is the responsibility of the creator. Without them screen readers won't be able to detect the elements.

Another notable argument against the need for web components is that the same functionality and pluggability can already be achieved with including JavaScript files. However, with traditional JavaScript includes you lose the encapsulation and interoperability promises.

At the time of this writing the browser support for Web Components is still lacking in the major browsers. Only Google Chrome and Opera support web components natively. Microsoft's Edge has only implemented templates and the other three technologies are still under development [9]. As stated before, Mozilla Firefox has agreed to not implement native web components at this point [8].

However, webcomponents.org [5] provides a comprehensive JavaScript polyfill which provides the

[5] webcomponents.org/polyfills

full functionality of Web Components in browser that do not support the standards natively. The existence of this polyfill is also one of the reasons Mozilla states that their decision to not ship native support is not harmful for the web community.

5 PRACTICAL EXAMPLES

In this chapter we demonstrate some practical use case examples. Throughout chapters 5.1 to 5.3 we demonstrate how different technologies incrementally benefit in single same use scenario. In chapter 5.4 we introduce our live interactive web demo which displays how some Web Components work live on browser.

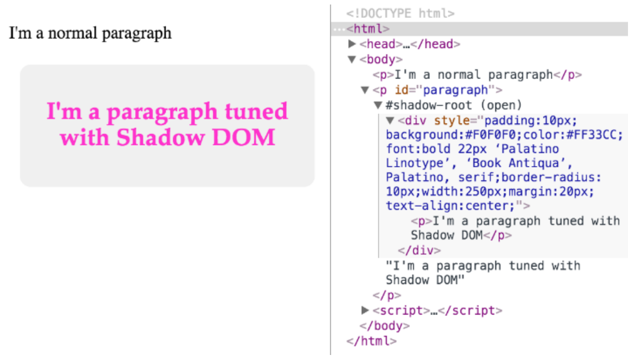
The examples in chapters 5.1 and 5.2 are also implemented into the live demo introduced in chapter 5.4.

5.1 SHADOW DOM

The following example code shows how Shadow DOM may be utilized to encapsulate an element to apply custom styling. Of the two paragraphs in document, the latter is encapsulated inside a shadow root and a div element to apply custom styling.

```
<p>I'm a normal paragraph</p>
<p id="shadow">I'm a paragraph tuned with Shadow
DOM</p>
<script>
  // Create shadow root on paragraph id='shadow':
  var host = document.querySelector('#shadow');
  var shadow = host.createShadowRoot();

  // Replicate the paragraph:
  var replicate = document.createElement('p');
  replicate.textContent = host.textContent;

  // Create a new div element with CSS styling:
  var newElement = document.createElement('div');
  newElement.setAttribute("style",
    "[...whatever styling you want...]");

  // Append the replicate into the new div element:
  newElement.appendChild(replicate);

  // Append the new div element to shadow root:
  shadow.appendChild(newElement);
</script>
```

Figure 2: Shadow DOM code example



I'm a normal paragraph

I'm a paragraph tuned with Shadow DOM

```
<!DOCTYPE html>
<html>
  ▶<head>…</head>
  ▼<body>
      <p>I'm a normal paragraph</p>
    ▼<p id="paragraph">
      ▼#shadow-root (open)
        ▼<div style="padding:10px;
        background:#F0F0F0;color:#FF33CC;
        font:bold 22px 'Palatino
        Linotype', 'Book Antiqua',
        Palatino, serif;border-radius:
        10px;width:250px;margin:20px;
        text-align:center;">
            <p>I'm a paragraph tuned with
            Shadow DOM</p>
          </div>
        "I'm a paragraph tuned with
        Shadow DOM"
      </p>
    ▶<script>…</script>
    </body>
</html>
```

Figure 3: Result

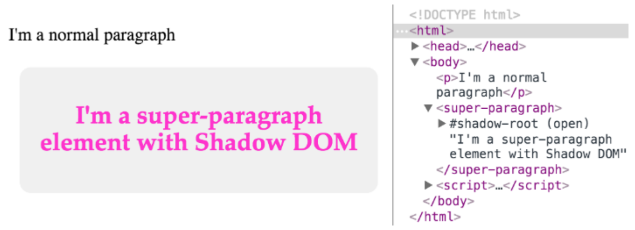
In the browser's developer tool we can peek inside the shadow root which is otherwise unavailable in document's DOM structure or scripts. We can see that the elements we created are there.

5.2 CUSTOM ELEMENT

We could make the previous example in chapter 5.1 more sophisticated by creating our custom element where the same shadow root styling is applied to content.

```
<p>I'm a normal paragraph</p>
<super-paragraph>I'm a super-paragraph element with
Shadow DOM</super-paragraph>
<script>
  // Create a new prototype based on 'HTMLElement'
  var superParagraphProto =
    Object.create(HTMLElement.prototype);

  // Callback for customizing prototype after instance
  // inserted into DOM
  superParagraphProto.createdCallback = function() {
    // Create shadow root and copy element content with
    // styling
    var shadow = this.createShadowRoot();
    // [... continues as in previous example ...]
  };

  // Declare new custom element using the customized
  // prototype
  var superParagraph =
    document.registerElement('super-paragraph', {
      prototype: superParagraphProto
    });
</script>
```

Figure 4: Custom Element code example



I'm a normal paragraph

I'm a super-paragraph element with Shadow DOM

```
<!DOCTYPE html>
<html>
  ▶<head>…</head>
  ▼<body>
      <p>I'm a normal
      paragraph</p>
    ▼<super-paragraph>
      ▶#shadow-root (open)
        "I'm a super-paragraph
        element with Shadow DOM"
      </super-paragraph>
    ▶<script>…</script>
    </body>
</html>
```

Figure 5: Result

5.3 HTML TEMPLATE AND IMPORT

To complete the examples, we will build the same example scenario as in chapters 5.1 and 5.2, utilizing the HTML Template and Import features. In our current implementation we create the shadow root's element structure with JavaScript manipulation methods, but we could also use HTML Template for that. Also we may put our element related script and styling markup to an external HTML file and import them into our main document. The following example shows how would the two html files (*index.html* and *import.html*) look.

```
<template id="superTemplate">
  <div>
    <p>
      <content></content>
    </p>
  </div>
  <style>
    div { [... styling for the div ...] }
  </style>
</template>

<script>
  // importDoc references this import's document
  var importDoc = document.currentScript.ownerDocument;

  // Create a new prototype based on 'HTMLElement'
  var superParagraphProto =
    Object.create(HTMLElement.prototype);

  // Callback for customizing prototype after instance
  // inserted into DOM
  superParagraphProto.createdCallback = function() {
    // get template in import
    var template =
      importDoc.querySelector('#superTemplate');

    // Make instance of the template by copying
    // template content
    var clone =
      document.importNode(template.content, true);

    var shadow = this.createShadowRoot();
    shadow.appendChild(clone);
  };

  // Declare new custom element using the customized
  // prototype
  var superParagraph =
    document.registerElement('super-paragraph', {
      prototype: superParagraphProto
    });
</script>
```

Figure 6: import.html

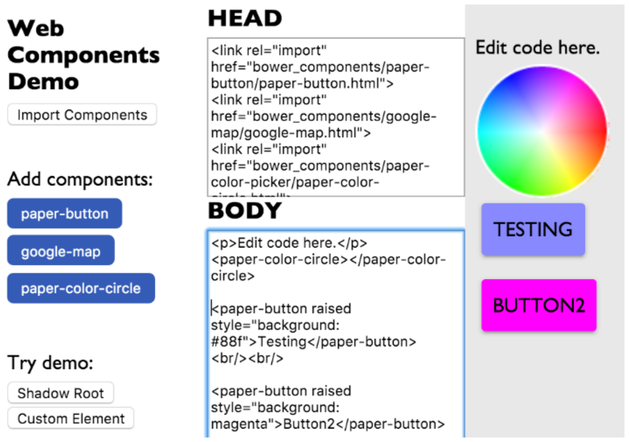
```
<!DOCTYPE html>
<html>
<head>
  <link rel="import" href="import.html">
</head>
<body>
  <p>I'm a normal paragraph</p>
  <super-paragraph>I'm a super-paragraph element with
Shadow DOM</super-paragraph>
</body>
</html>
```

Figure 7: index.html

The end result looks similar as in previous example in chapter 5.2. The key difference here is the cleaner syntax for defining the shadow root structure and the neat separation of content and functionality. The index.html file is very minimal and contains only well comprehensive syntax and actual content.

5.4 LIVE DEMO

We implemented a live demo application to demonstrate the Web Component usage in a real browser. The application is targeted for developers who are interested in how do the Web Components work in practice. The live demo is hosted on web for easy access[6] and the source code for the demo is available on GitHub[7].



```
<link rel="import"
href="bower_components/paper-button/paper-button.html">
<link rel="import"
href="bower_components/google-map/google-map.html">
<link rel="import"
href="bower_components/paper-color-picker/paper-color-
```

```
<p>Edit code here.</p>
<paper-color-circle></paper-color-circle>

<paper-button raised style="background: #88f">Testing</paper-button>
<br/><br/>

<paper-button raised style="background: magenta">Button2</paper-button>
```

Figure 8: Screenshot of the live demo app

The demo consists of three columns. On the middle column the "head" text field displays what imports have been included in the web page DOM. The lower "body" text field displays the web page body. On the right column is a live render of the middle columns' code. The user may code HTML into the "body" text field and it is soon rendered to the right column.

On the left column there are buttons for adding ready Web Component implementations to the coding "body" field. The first button "import elements" displays a dialogue that enables the user to add more imports to the page. The added imports are displayed in the "head" text field. The lower buttons add enabled Web Component implementations to the "body" field to demonstrate their usage.

The lowest two buttons demonstrate the examples presented in chapters 5.1 and 5.2. The example's JavaScript code is loaded into the "head" text field and example's body elements are appended to the "body" field. Then the corresponding live render of the example is rendered on the right column.

6 CONCLUSIONS

Web Components are a set of standardized native browser technologies. They are a future-proof technology that solves the interoperability issues caused by traditional JavaScript framework. The standards encourage semantic syntax and modular thinking. The encapsulation allows the developer to import new components onto a website without style or script overlap.

At the time of writing the standards are not yet final and are subject to changes in the future. They are not yet utilized widely in the industry and the browser support is not complete in the major browsers. However,

[6] http://users.aalto.fi/hellec1/webcomponents/

[7] https://github.com/chriisu/webcomponents-demo/

the solid polyfill library enables the use of the standards in all major browsers. Because of this we conclude that Web Components are already a mature set of technologies ready for production use.

7 Acknowledgments

This paper is course work for the “T-111.5360 - WWW Applications P” –course in Aalto University in the fall of 2015 by Petri Vuorimaa.

8 References

[1] T. Savage, "Componentizing the web," *Communications of the ACM,* vol. 58, no. 11, pp. 55-61, 23 10 2015.

[2] D. Glazkov, "Custom Elements W3C Working Draft," 16 12 2014. [Online]. Available: http://www.w3.org/TR/2014/WD-custom-elements-20141216/.

[3] D. Glazkov, R. Weinstein and T. Ross, "HTML5 A vocabulary and associated APIs for HTML and XHTML W3C Recommendation," 28 10 2014. [Online]. Available: http://www.w3.org/TR/2014/REC-html5-20141028/.

[4] D. Glazkov and H. Ito, "Shadow DOM W3C Working Draft," 6 10 2015. [Online]. Available: http://www.w3.org/TR/2015/WD-shadow-dom-20151006/.

[5] G. Dimitri and M. Hajime, "HTML Imports W3C Working Draft," 11 3 2014. [Online]. Available: http://www.w3.org/TR/2014/WD-html-imports-20140311/.

[6] W. He, D. Akhawe, S. Jain, E. Shi and D. Song, "ShadowCrypt: Encrypted Web Applications for Everyone," in *2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[7] A. Van Kesteren, "Mozilla Hacks," 15 12 2014. [Online]. Available: https://hacks.mozilla.org/2014/12/mozilla-and-web-components/. [Accessed 10 12 2015].

[8] J. Peek, "Webcomponents.org," 29 9 2014. [Online]. Available: http://webcomponents.org/articles/interview-with-joshua-peek/. [Accessed 17 11 2015].

[9] Microsoft Edge Team, "Microsoft Edge Dev Blog," 15 07 2015. [Online]. Available: https://blogs.windows.com/msedgedev/2015/07/15/microsoft-edge-and-web-components/. [Accessed 10 12 2015].